Algorithm: Lab3 (By Sujiv Shrestha ID:610145)

Problem 1.

1. Goofy has thought of a new way to sort an array arr of n distinct integers:
    a. Step 1: Check if arr is sorted. If so, return.
    b. Step 2: Randomly arrange the elements of arr (Hint: this can be done in O(n))
    c. Step 3: Repeat Steps 1 and 2 until there is a return.

Answer the following:

A. Will Goofy's sorting procedure work at all? Explain

Ans: Such a process of sorting by Hit and Trial method is all based upon chances or probability. The process will surely work for atleast once during infinite attempts but it is quite unpredictable how many iterations are required. The probability of getting success (sorted array) decreases as the length of array increases.

B. What is a best case for GoofySort?

Ans: The best case is when the input array "arr" is already sorted.

C. What is the running time in the best case?

Ans: The running time in the best case is $\Theta(n)$ where n is the length of the input array as there are n comparisons required to check if array is sorted.

D. What is the worst-case running time?

Ans: The worst-case running time is undefined as it is all based upon chances.

E. Is the algorithm inversion-bound?

Ans: Yes, the algorithm is inversion-bound because it makes at least as many comparisons as there are inversions on any input array.

Problem2

2. *Interview Question.* An array A holds n integers, and all integers in A belong to the set {0, 1, 2}. Describe an O(n) sorting algorithm for putting A in sorted order. Your algorithm may not make use of auxiliary storage such as arrays or hashtables (more precisely, the only additional space used, beyond the given array, is O(1)). Give an argument to explain why your algorithm runs in O(n) time.

```
Algorithm sort012(A,n)
    Input array A of n integers containing integers from set{0,1,2}
    Output array A in sorted order
    indexZero←0
    indexOne←0
    for i← 1 to n do
        if A[i] = 0 then
            temp←A[indexZero]
            A[indexZero] ←A[i]
            A[i] ←A[indexZero+indexOne]
            A[indexZero+indexOne] ←temp
```

```
            indexZero←indexZero+1
        else if A[i] = 1 then
            temp←A[i]
            A[indexZero+indexOne] ←A[i]
            A[i] ←temp
            indexOne←indexOne+1
    return C
```

Java implementation of the above algorithm:

```
    public int[] sort012(int[] A) {
        int ind0 = 0;
        int ind1 = 0;
        int temp = 0;
        for(int i=0;i<A.length;i++) {
            if(A[i]==0) {
                temp = A[ind0];
                A[ind0] = A[i];
                A[i] = A[ind0+ind1];
                A[ind0+ind1] = temp;
                ind0++;
            }
            else if(A[i]==1) {
                temp = A[ind0+ind1];
                A[ind0+ind1]=A[i];
                A[i] = temp;
                ind1++;
            }
            //System.out.println(Arrays.toString(A));
        }
        return A;
    }
```

The above algorithm sorts the array containing integers from set{0,1,2} in O(n) because we know the order of the integers in the set and by the use of index to the last 0's and 1's we can exactly define where should be the new element in the integer should be swapped with. Hence it is sorted in O(n).

Following example shows the steps involved in sorting test input `[0, 2, 2, 1, 1, 0, 2, 1, 0]`

```
Iteration 1: [0, 2, 2, 1, 1, 0, 2, 1, 0]
Iteration 2: [0, 2, 2, 1, 1, 0, 2, 1, 0]
Iteration 3: [0, 2, 2, 1, 1, 0, 2, 1, 0]
Iteration 4: [0, 1, 2, 2, 1, 0, 2, 1, 0]
Iteration 5: [0, 1, 1, 2, 2, 0, 2, 1, 0]
Iteration 6: [0, 0, 1, 1, 2, 2, 2, 1, 0]
Iteration 7: [0, 0, 1, 1, 2, 2, 2, 1, 0]
Iteration 8: [0, 0, 1, 1, 1, 2, 2, 2, 0]
Iteration 9: [0, 0, 0, 1, 1, 1, 2, 2, 2]
```

Note: The swapped elements are denoted as red.

Problem3

3. *BubbleSort*

a. Improve the BubbleSort implementation so that when the input array becomes sorted after some runs of outer for loop, the algorithm will stop. Call your new Java file BubbleSort1.java.

b. Recall that in BubbleSort, at the end of the first pass through the outer loop, the largest element of the array is in its final sorted position. After the next pass, the next largest element is in its final sorted position. After the *i*th pass (i=0,1,2,…), the largest, second largest,…, i+1st largest elements are in their final sorted position. Use this observation to cut the running time of BubbleSort in half. Implement your solution in code, and prove that you have improved the running time in this way. Call your new Java file, which contains the improvements from this problem and the previous problem, BubbleSort2.java.

c. In this lab folder, I have given you an environment for testing sorting routines. Insert into this environment the original BubbleSort file along with your new BubbleSort1 and BubbleSort2 classes, and run the SortTester class. What are the results? Are the results what you expected? Explain why the running times turned out the way they did.

Answer: After making modification as per instruction from a. it was observed that the running was drastically reduced and it ranked 1st instead of 3rd.

```
2 ms -> BubbleSort1
271 ms -> InsertionSort
449 ms -> SelectionSort
1596 ms -> BubbleSort
```

One reason was observed that most of the elements in the input array was already in sorted order. So, the array got sorted in few iterations.
While after the additional modification as per instruction from b. was supposed to reduce the running time to half as the second for loop runs from 0 to (n-1-i) reducing the number of iterations from n*n to n(n+1)/2 i.e from $O(n^2)$ to $O(n^2/2)$. But the result showed.

```
2 ms -> BubbleSort1
2 ms -> BubbleSort2
271 ms -> InsertionSort
449 ms -> SelectionSort
1596 ms -> BubbleSort
```

Again the reason for this might be the input array might have been sorted in just few outer loops so that the significance of second modification was not noticeable.

Problem4

4. *Interview Question.* You are given a length-n array A consisting of 0s and 1s, arranged in sorted order. Give an o(n) algorithm that counts the total number of 0s and 1s in the array. Your algorithm may not make use of auxiliary storage such as arrays or hashtables (more precisely, the only additional space used, beyond the given array, is O(1)). You must give an argument to show that your algorithm runs in o(n) time.

```
Algorithm count01 (A,n)
    Input array A of n integers containing integers 0's and 1's
    Output count of zeros
    for i← 1 to n do
        if A[i] = 1 then
            return i        //count of one will be obviously understood as n – count of zeros.
```

Explanation:

The algorithm above gives the count of 0 for the sorted input array of 0's ant 1's in o(n) because it only needs to find the position where the sequence of 0's end.

So, the best case is when the there are no zeros. Only single comparison can tell how many zero's and one's are there. For this condition the number of zero's will be zero and number of one's will be equal to the length of array.

Similarly, the worst case is when there are only zero's. In this case the loop runs till the end to check if there are any one's in the array.